VerLog: Enhancing Release Note Generation for Android Apps using Large Language Models

JIAWEI GUO, University at Buffalo, SUNY, USA
HAORAN YANG, Washington State University, USA
HAIPENG CAI*, University at Buffalo, SUNY, USA

Release notes are essential documents that communicate the details of software updates to users and developers, yet their generation remains a time-consuming and error-prone process. In this paper, we present VerLog, a novel technique that enhances the generation of software release notes using Large Language Models (LLMs). VerLog leverages few-shot in-context learning with adaptive prompting to facilitate the graph reasoning capabilities of LLMs, enabling them to accurately interpret and document the semantic information of code changes. Additionally, VerLog incorporates multi-granularity information, including fine-grained code modifications and high-level non-code artifacts, to guide the generation process and ensure comprehensive, accurate, and readable release notes. We applied VerLog to the 42 releases of 248 unique Android applications and conducted extensive evaluations. Our results demonstrate that VerLog significantly (up to 18%–21% higher precision, recall, and F1) outperforms state-of-the-art baselines in terms of completeness, accuracy, readability, and overall quality of the generated release notes, in both controlled experiments with high-quality reference release notes and in-the-wild evaluations.

CCS Concepts: • **Software and its engineering → Documentation**.

Additional Key Words and Phrases: LLMs, release notes, automated change documentation, Android apps

ACM Reference Format:
Jiawei Guo, Haoran Yang, and Haipeng Cai. 2025. VerLog: Enhancing Release Note Generation for Android Apps using Large Language Models. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA084 (July 2025), 23 pages. https://doi.org/10.1145/3728961

1 Introduction

Software development is a dynamic and iterative process, with applications constantly evolving to meet user needs, fix bugs, and introduce new features [40]. In this ever-changing landscape, release notes serve as a crucial bridge between developers and users, documenting the journey of software from one version to the next [8]. These concise yet informative documents play a pivotal role in the software lifecycle [2], offering a snapshot of what has changed and why [8, 41].

The importance of release notes extends far beyond a simple changelog [2]. For *users*, they provide a roadmap to navigate new features, understand resolved issues, and adapt to changes in functionality [46]. A well-crafted release note can enhance user experience, reduce support requests, and foster a sense of transparency and trust between the software provider and its community. For

*Haipeng Cai is the corresponding author.

Authors' Contact Information: Jiawei Guo, University at Buffalo, SUNY, Buffalo, USA, jiaweigu@buffalo.edu; Haoran Yang, Washington State University, Pullman, USA, haoran.yang2@wsu.edu; Haipeng Cai, University at Buffalo, SUNY, Buffalo, USA, haipengc@buffalo.edu.

ACM 2994-970X/2025/7-ARTISSTA084
https://doi.org/10.1145/3728961

developers, release notes serve as a historical record, facilitating collaboration across teams and providing context for future modifications [41] hence potentially mitigating evolution-induced issues such as application incompatibilities [10, 19, 50]. They are also invaluable for *project managers*, offering insights into the pace of development and the focus of recent efforts [4].

Despite their importance, the generation of release notes is far from being trivial [4, 8]. Intuitively, generating comprehensive, accurate, and useful release notes requires a deep understanding of the changes made, the ability to distill complex technical modifications into user-friendly language, and the foresight to anticipate what information will be most valuable to the audience [41]. *Manually writing release notes* is a common option, yet it is often a labor-intensive and expensive process [31], which may lead to low-quality notes. Meanwhile, other forms of change documentation, such as commit messages and change logs, themselves may not capture the level of details and comprehensiveness expected of a quality release note [2, 23]. This task becomes increasingly daunting as software projects grow in complexity and the frequency of releases accelerates in the continuous/agile development era.

Recognizing these hurdles, researchers attempted to automate release note generation [4]. Almost all of the existing *automated methods* rely solely on the reorganization, summarization, or categorization of *non-code* (textual) artifacts such as commit messages, pull requests, and issues [21, 23, 33, 35]. While this approach can streamline the process, it is inherently limited by the quality and comprehensiveness of these artifacts. For instance, commit messages may be poorly written [22, 43, 49], issues vaguely described [51], and/or pull requests inconsistently documented [28, 44]. As a result, release notes generated from these non-code artifacts suffer in quality. Moreover, software projects with limited community engagement may lack the rich ecosystem of issues and pull requests that more popular projects enjoy. Figure 1 shows an example which illustrates the limitation.

Release Notes generated for app cityfreqs.com.pilfershushjammer v4.6.0

Ground Truth	TextRank	Github Changelog Generator	VerLog
- min API bump to 21 (5.0, Lollipop) - github branch archive2 prior to min api 21 - bump - Android 11 add foregroundServiceType - add intent for browser package access - prototyping Assistant/Omnibox jammer - add excessive retrigger warning - catch MediaRecorder placebo exception in - some devices - update Android Studio - add SDK name - buildVersion 41	release version additions for assistant jamming note for Android 12 add another SDK name add SDK name placebo exception handling integrate Voice Assistant jammer Merge pull request #32 from c-4-m/master Update German Translation excessive warning note for toast option voice settings what voice assist jamming api30 testing markdown syntax.	**Closed issues:** - German Translation [\#26] (https://github.com/kaputnikGo/PilferShushJammer/issues/26) - Option to start application on boot [\#21] (https://github.com/kaputnikGo/PilferShushJammer/issues/21) **Merged pull requests:** - Update and improve German translation [\#32] (https://github.com/kaputnikGo/PilferShushJammer/pull/32) ([c-4-m](https://github.com/c-4-m))	- Integrated a Voice Assistant jammer to block unwanted access to the onboard microphone, - Added a new option to open the project's GitHub page for more information. - Introduced a feature that provides users with a notification when voice assistance is blocked - Implemented excessive warning notifications to alert users when another app frequently attempts to access the microphone - Updated German Translation for improved user accessibility.

Fig. 1. Release notes written by developer ("ground truth") and generated by existing, textual-based tools (TextRank and Changelog Generator) versus VerLog.

Figure 1 presents our comparative analysis of an app `cityfreqs.com.pilfershushjammer` [1] on version 4.6.0. The figure includes four sets of release notes: the ground truth (a manually crafted set of notes by the app developer), TextRank (an approach based on text summarization), and GitHub Changelog Generator (which relies on GitHub artifacts such as issues and pull requests) and VerLog. TextRank (yellow background) provides only a superficial summary, often missing the deeper context and specificity of feature updates. For example, TextRank's output contains generic phrases that fail to convey the precise functional changes in the software. The GitHub Changelog Generator (blue background) is similarly limited by its dependency on external documentation, leading to formal but incomplete summaries when issues and pull requests are lacking or inconsistently documented. As a result, it fails to capture the nuanced changes in projects with sparse or poorly maintained metadata. It is also worthwhile noticing that, the ground truth (pink background), i.e., release notes written by developer, is not without drawbacks. Although it might have covered all relevant changes, it often lacks organization and contains excessive developer-focused, low-level details that may overwhelm end-users. This clutter can make it challenging for users to quickly grasp the key improvements or bug fixes, as the note is tailored more to developers than to the general audience. These limitations underscore the inherent challenges in

generating high-quality release notes from non-code artifacts alone, particularly for projects with limited commit messages or inconsistent community contributions.

While approaches using only non-code artifacts have the merit of being language-agnostic [21, 33], a robust and accurate release note generation technique should analyze and incorporate code-level changes—after all, the truth about software behavior lie in the code itself. Yet such techniques are rare, represented by ARENA [31]. It summarizes code changes identified from commits and links the summaries to information in commit notes and issue trackers. However, the code summarization does not capture code semantics well, while the reliance on the non-code information makes it suffer the respective drawbacks discussed earlier. Its quantitative effectiveness also remains unknown.

A more desirable approach would minimize dependence on potentially unreliable non-code/textual artifacts, while capturing the semantic essence of code changes. In recent years, Large Language Models (LLMs) exhibit remarkable potential in both of these tasks [11, 32]. Leveraging these advanced models presents an opportunity to automate and enhance the process of release note generation. However, as we show in our results, straightforward use of LLMs falls short in capturing the nuanced and structured information required for high-quality release notes, particularly in understanding the semantic relationships between code changes, due to several major challenges. First (*Challenge-1*), low-level code changes alone do not carry enough semantic information for LLMs to understand feature-level intentions of code changes without context, while providing more context pressurizes the limited token/context capacity of LLMs. Second (*Challenge-2*), those feature-level intentions are related to knowledge specific to the functionality *domain* of the software, which may be missing in LLMs. Third (*Challenge-3*), using coarser-grained (e.g., method-level) change information can reduce the pressure on LLMs' token capacity, but such information alone may not be sufficient for LLMs to accurately comprehend subtle changes in code semantics.

In this paper, we introduce VerLog, an LLM-based technique that aims to enhance automated generation of software release notes by combining *graph-based prompting* and *in-context learning* on LLMs, both focused on code changes, while utilizing *multi-granularity information* about those changes. Given a software release, VerLog first identifies method-level code diffs and prompts the LLM to reason about graph-represented knowledge on the semantics of these diffs, according to the minimum subgraph of the program call graph that covers all the changed methods and is minimally contextualized with unchanged ones. By including context that is essential for high-level semantics understanding with LLMs while minimizing this context to reduce token use and fit their limited context window, this design address *Challenge-1*. Next, in its few-shot in-context learning, VerLog adaptively selects prompting exemplars from a pre-curated pool according to the functionality domain of the software under analysis while covering diverse semantic categories of changes, which augments LLMs with necessary domain knowledge hence addresses *Challenge-2*. Finally, in addition to the graph knowledge and exemplars, VerLog includes in the final prompt various other information at multiple granularity levels: statement-level code diffs aligned with their associated method-level diffs, commit messages, and project description, addressing *Challenge-3*.

We have implemented VerLog for Android apps given their broad impact and large user base [9], and applied it to 248 releases across 42 unique popular apps. Given the lack of benchmarks with high-quality release notes, we curated one and used it for our evaluation. Against these notes as the ground truth, VerLog achieved substantially greater effectiveness (up to 18% higher precision, 19% higher recall, and 21% higher F1, respectively) than the state-of-the-art release note generators both in academia and industry. Our extensive ablation studies further show that VerLog retains its significant superiority even without using any non-code artifacts. In terms of efficiency, VerLog produces a release note in about 30 seconds on average. For an in-the-wild evaluation via a user study, VerLog generates high-quality release notes for 10 apps whose original notes are poor, with

an average score of 3.81, 3.63, 4.56, and 3.94 in terms of completeness, accuracy, readability, and overall quality based on participants' Likert ratings ranging from 1 for lowest and 5 the highest.

Through our further inspection of the cases in our in-the-wild evaluation, we found that existing release notes, even of the highly popular and impactful apps, are often of quite poor quality (e.g., missing a note for important changes, obscure description of changes captured, and low readability overall). Given the high manual cost of writing high-quality release notes, VerLog can play a major role in enhancing developer productivity in such a critical software documentation task.

The key contributions of this paper include:

- The first technique leveraging static code analysis and LLMs for release note generation, departing from traditional approaches which rely solely on textual artifacts or information retrieval.
- A novel approach to automated release note generation that exploits LLMs' graph reasoning to capture code-change semantics and augments them with functionality domain knowledge via adaptive LLM prompting, while leveraging multi-granularity information (§2).
- An open-source tool that implements VerLog for Android apps and a curated benchmark of high-quality release notes for such apps (§3.1).
- Extensive evaluations of VerLog, including controlled and in-the-wild experiments, that demonstrates it substantial merits over research and industry state-of-the-art peer tools while revealing major issues with real-world release notes of popular/impactful apps (§3.3).

2 Methodology

VerLog is designed to enhance the generation of software release notes by leveraging the power of Large Language Models (LLMs) in conjunction with sophisticated code analysis techniques. Our approach addresses the key challenges in automated release note generation by incorporating semantic understanding of code changes, adaptive learning, and multi-granularity information processing. Figure 2 presents an overview of VerLog's architecture and workflow.

① **Graph Knowledge Based Prompting**. This component forms the foundation of VerLog's semantic understanding capabilities. It constructs a partial call graph representation of the code changes between two software versions. By encoding the changed methods and their relationships as a graph, we enable the LLM to reason about the semantic context of the modifications, going beyond simple textual diffs. As shown in Figure 2, this component corresponds to *Phase 1* of the whole workflow. It will take the base app and release app as input, and then find changed methods, and then extract their callees from corresponding call graphs. With these obtained, we can slice a subgraph which can be change-aware.

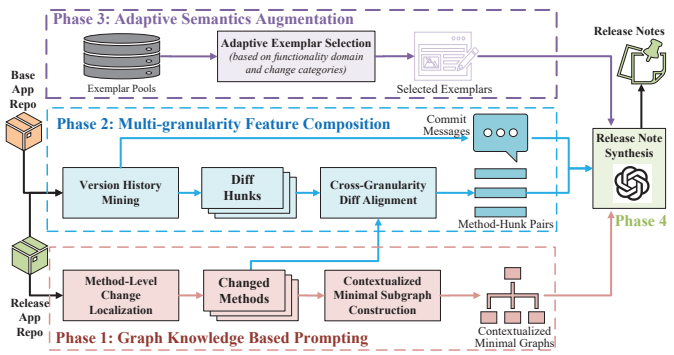

Fig. 2. Overview of VerLog architecture and workflow.

② **Multi-granularity Feature Composition**. To provide a comprehensive view of the changes, this module integrates information at various levels of granularity. It aligns fine-grained code modifications with method-level changes and incorporates high-level information such as commit messages and application descriptions. This multi-faceted approach ensures that both detailed code changes and overarching project context are considered in the release note generation process. In Figure 2, this part corresponds to

Phase 2. The version history will be utilized to obtain diff hunks ,which will be further aligned with changed methods obtained in *Phase 1* to enrich the prompt.

③ **Adaptive Semantics Augmentation**. This component enhances the LLM's ability to generate contextually appropriate release notes through few-shot in-context learning. It curates and selects exemplars based on the specific category of the application and the types of code changes, allowing the model to adapt its output to different software domains and change patterns. This component corresponds to *Phase 3* in Figure 2.

④ **In-Context Learning Based Feature Change Summarization**. Leveraging the prepared prompts and selected exemplars, this module utilizes the LLM to generate summaries of the code changes. The LLM interprets the graph-based code representations, multi-granularity information, and adaptive exemplars to produce detailed yet coherent descriptions of the modifications. As shown in Figure 2, the LLM will infer and summarize from the prompt which is the combination of the output of all three phases.

⑤ **Summarization Synthesis**. The final component in the pipeline consolidates the individual change summaries into a cohesive and well-structured release note. It addresses challenges such as redundancy elimination, abstraction of low-level details, and logical organization of the information to produce a user-friendly final document.

2.1 Graph Knowledge Based Prompting (Phase 1)

LLMs are known to be capable of graph reasoning tasks represented in natural language [45] [42] [7]. We leverage this capability through a graph-based approach that serves two critical purposes. First, by analyzing the call graph and extracting its contextualized minimal subgraph (CMG) containing changed methods, we can naturally partition code changes into semantically coherent groups, where each group becomes a separate prompt. Second, the extracted CMG provides rich semantic context about method relationships and interactions, enabling LLMs to better reason about the nature and impact of code changes.

Next, we elaborate on the two key steps of Phase 1: (1) identifying changed methods and (2) constructing the contextualized minimal subgraphs (CMGs) that enable both effective change grouping and semantic understanding.

2.1.1 Method-Level Change Location. We first identify changed methods via method-level differencing and the corresponding changed classes. With app source code available, this step is straightforward. Yet it may result in false positives due to code refactoring. We chose a lightweight approach to mollify it by only detecting added, modified, and deleted class files, specifying corresponding options provided by git [17].

After this differencing, we will obtain three multi-sets–added classes and their corresponding methods (these method are intrinsically added methods, similar for deleted ones), deleted classes and their corresponding deleted classes, and modified classes and corresponding three categories of changed methods (added, modified, and deleted methods). Along with each method's name and parameters, the line numbers of the start line and end line will also be marked for the method-hunk alignment in the next step.

2.1.2 Contextualized Minimal Subgraph (CMG) Construction. After obtaining changed methods, we construct a contextualized minimal subgraph (CMG) that captures both the changed methods and their semantic relationships in the codebase. The goal is to provide enough context for LLMs to understand not just what methods changed, but how they interact with and influence other parts of the system. For example, if a method is modified, understanding its callers and callees helps determine whether the change affects core functionality or is localized to implementation details.

To achieve this, we chose the call graph as the graph representation, which has been used by prior work to provide context to enhance code summarization [29] [6].

Callees Extraction. Here we refer a method's *context* as the setting in which the method is called [24]. For each release, we checked out to the corresponding commit and build the app without obfuscation. For each changed method, which is obtained from the previous step, we extract its callees, i.e., the immediate successors in the call graph for the context formation in the next step.

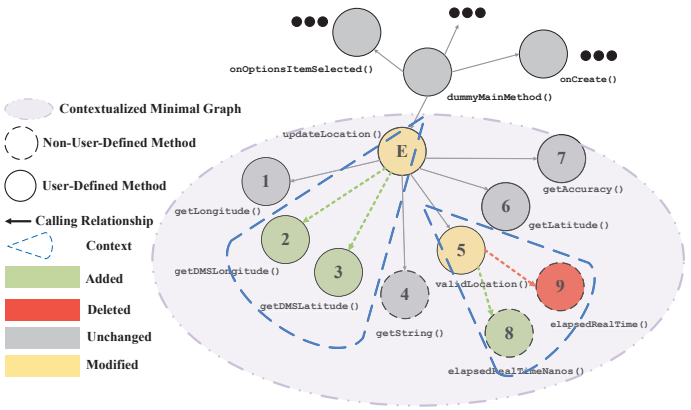

Fig. 3. An illustrating CMG of a real-world Android app (of Figure 4).

Context Formation. Compared to one single method, adding its callees can already provide some *context* for it. However, this *context* is not enough to infer the semantics of the method in some cases. For instance, if a method is added and called by another user-defined methods, simply using this added method and its caller as the context may be enough to infer the semantics of the added method, but will miss the semantics of the caller method, which is vital to understand the impact of the added method. Figure 3 illustrated this scenario, where two added methods `getDMSLongitude`(Node 2) and `getDMSLatitude`(Node 3). Their functionality can be understood even without callees. However, if their callers cannot be known(In this example, it is `updateLocation`, Node *E*), it would be impossible to infer the end-user perceivable functionality pertaining to this addition change. This calls for a solution especially when we are interested in finding the functionality feature changes in the new release. The *context* should be able to provide a comprehensive view of changes.

To construct a rich context, we link changed user-defined methods by examining their callees. We form a context when a callee of a changed method is also a changed method, including both methods and their callees. This process continues recursively until no more changed methods are found in the callees. The resulting context forms a tree structure where: (1) Root nodes are changed methods not called by other changed methods, (2) Non-leaf nodes are changed user-defined methods and (3) Leaf nodes are unchanged user-defined methods or external methods.

To elucidate the context formation, we provide an example in Figure 3 and present Algorithm 1. The algorithm processes two sets of changed methods: the Reference set R (old version) and the Target set T (new version), each containing methods and their respective callees. It outputs a set of change-aware contextualized subgraphs CS that capture the structural and semantic relationships between changes.

The algorithm first identifies entry methods - changed methods not called by other changed methods - which serve as natural starting points for analysis (lines 2-3). Using these entry points, it constructs subgraphs for both reference and target versions (lines 4-5), capturing the call relationships between changed methods and their surrounding context.

The core analysis phase (lines 7-19) compares and categorizes changes by examining each method's presence across versions. Methods present in both versions are merged to represent modifications (lines 10-11), while methods unique to the reference or target versions are marked as deleted (lines 12-14) or added (lines 15-17) respectively. This process creates a unified view that preserves both the individual changes and their relationships.

Algorithm 1: Change-aware Context Formation

```
Input : Reference set R, Target set T of changed methods
        and their callees
Output: Set of Change-aware contextualized subgraphs CS
1  CS ← ∅
2  R_entry ← FindEntryMethods(R)
3  T_entry ← FindEntryMethods(T)
4  R_subgraphs ← BuildSubgraphs(R_entry, R)
5  T_subgraphs ← BuildSubgraphs(T_entry, T)
6  all_roots ← {subgraph.root | subgraph ∈
     R_subgraphs ∪ T_subgraphs}
7  foreach root ∈ all_roots do
8      r_subgraph ← FindSubgraph(R_subgraphs, root)
9      t_subgraph ← FindSubgraph(T_subgraphs, root)
10     if r_subgraph ≠ ∅ ∧ t_subgraph ≠ ∅ then
11         merged_subgraph ←
             MergeSubgraphs(r_subgraph, t_subgraph)
           CS ← CS ∪ {merged_subgraph}
12     else if r_subgraph ≠ ∅ then
13         r_subgraph.root.label ← " − "
14         CS ← CS ∪ {r_subgraph}
15     else
16         t_subgraph.root.label ← " + "
17         CS ← CS ∪ {t_subgraph}
18     end
19 end
20 return CS
21 Function FindEntryMethods(MethodSet):
22     all_methods ← {m.name | m ∈ MethodSet}
23     reachable ← ⋃_{m∈MethodSet} m.callees
24     return
         {m | m ∈ MethodSet ∧ m.name ∉ reachable}
25 Function BuildSubgraphs(EntryMethods, AllMethods):
26     subgraphs ← ∅
27     foreach entry ∈ EntryMethods do
28         subgraph ←
             BuildSubgraphRecursive(entry, AllMethods)
           subgraphs ← subgraphs ∪ {subgraph}
29     end
30     return subgraphs
31 Function MergeSubgraphs(Subgraph1, Subgraph2):
32     merged ←
         MergeNodes(Subgraph1.root, Subgraph2.root)
33     return CreateSubgraph(merged)
```

The final output, a set of Change-aware CMGs (line 20), offers a rich, structured representation of how changes in one part of the system relate to and influence others, enabling more accurate reasoning about the impact and scope of modifications.

Graph Representation. Given that we have obtained the CMGs, the next step is to represent them in a way that can be easily understood by the LLM. To achieve this, we adopted the graph representation proposed by prior work [42], which has been shown to boost the performance of LLM in graph reasoning tasks.

This representation will give the LLM both nodes and edges of the partial call graph separately. Each node will be represented by the method signature and a prefix number as an index. The index, instead of the method signature, will be used in the edge representation. This representation, as per the prior work [42], can not only help to overcome the limitation of LLM understanding graph structure in pure textual form but also reduce the token used for each prompt. Figure 4 shows an example of this representation, i.e., how a CMG is encoded in the prompt. This prompt corresponds to the CMG in Figure 3. The entry method node E in the CMG corresponds to line 2 in the prompt, whereas Nodes 1—9 correspond to the same label of method signature starting from line 27. The edges are shown in the bottom of the prompt, starting from line 73.

2.2 Multi-Granularity Feature Composition (Phase 2)

We enhance our LLM prompts by incorporating fine-grained code changes alongside method-level context. This approach leverages developers' full source code access, enabling the LLM to capture detailed semantic changes. Additionally, we include commit messages, drawing from version history research. This multi-layered input—combining coarse and fine-grained changes with commit insights—allows the LLM to generate more accurate and comprehensive release notes while balancing information richness with LLM capacity constraints.

2.2.1 Cross-Granularity Diff Alignment. To incorporate fine-grained code changes, we developed a heuristic algorithm for hunk-method alignment in diff files. This approach addresses the limitations of existing tools that struggle with aggregating changes across multiple commits between releases.

Our algorithm categorizes alignments into three cases: one hunk to one method, one hunk to multiple methods, and multiple hunks to one method. Considering the three types of method changes (added, modified, deleted), we identified nine possible alignment scenarios, with two being practically impossible. (One added/deleted method matches with multiple hunks)

Algorithm 2: Method-Hunk Alignment

```
   Input  : Method M, Change Category C, Set of Hunks H
   Output : Set of Matched Lines L
1  L ← ∅
2  foreach hunk h ∈ H do
3      if C = MODIFIED then
4          L ← L ∪ MatchModifiedMethod(M, h)
5      else if C = ADDED then
6          L ← L ∪ MatchAddedMethod(M, h)
7      else if C = DELETED then
8          L ← L ∪ MatchDeletedMethod(M, h)
9      end
10 end
11 return L
12 Function MatchModifiedMethod(M, h):
13     start, end ← GetMethodLines(M)
14     if h.targetStart ≤ start ∧ end ≤ h.targetEnd then
15         return ExtractMethodLines(h, M)
16     else if start ≤ h.targetStart ≤ end then
17         return
             ExtractPartialMethodLines(h, M, start, end)
18     else
19         return ∅
20     end
21 Function MatchAddedMethod(M, h):
22     start, end ← GetMethodLines(M)
23     if h.targetStart ≤ start ∧ end ≤ h.targetEnd then
24         return ExtractMethodLines(h, M)
25     else
26         return ∅
27     end
28 Function MatchDeletedMethod(M, h):
29     start, end ← GetMethodLines(M)
30     if h.sourceStart ≤ start ∧ end ≤ h.sourceEnd then
31         return ExtractMethodLines(h, M)
32     else
33         return ∅
34     end
```

The algorithm processes four key hunk attributes: target start line, target length, source start line, and source length. It determines the quantitative relationship between these attributes and the method's start and end lines. For instance, an added method is aligned with a hunk if the hunk's target range fully encompasses the method's lines.

To illustrate the method-hunk alignment process, we present Algorithm 2. The algorithm aligns method changes with their corresponding diff hunks by taking a method M, its change category C (MODIFIED, ADDED, or DELETED), and a set of git diff hunks H as input. It outputs a set of matched lines L that precisely capture the changes specific to the input method.

The algorithm matches each hunk (lines 2-10) based on the method's change category. For modified methods (lines 3-4), it identifies overlapping regions between the method's boundaries and the hunk, handling both full and partial containment cases. For added methods (lines 5-6), it checks if the method appears entirely within the hunk's target lines, while for deleted methods (lines 7-8), it examines the hunk's source lines.

Each matching considers four hunk attributes: target start line, target length, source start line, and source length. By comparing them with the method's start and end lines, the algorithm determines the precise portions of the hunk that correspond to the method's changes. The matched lines from all relevant hunks are accumulated (lines 4, 6, and 8) to create a complete representation of the method's modifications. In Figure 4, lines marked in blue correspond to the matched diff hunks.

2.2.2 Feature Summarization Enhancement. To enhance the LLM's understanding of changes and the application context, we incorporate system and package-level information into our prompt. Specifically, we include commit messages and application descriptions. Commit messages serve as valuable reference material for the LLM. They help verify the summarization results and often provide insights into system or package-level changes that may not be evident from code changes alone, such as policy or license updates. This additional context ensures that important non-code modifications are captured in the generated release notes. Application descriptions offer a high-level overview of the app's functionality, target audience, and key features. By including this information, we enable the LLM to contextualize code changes within the broader scope of the application's purpose and user expectations. This comprehensive understanding allows the LLM to generate release notes that are more aligned with user interests, highlighting changes that are most relevant and impactful from a user perspective. In Figure 4, the blue area that covers line 1 to line 8 belongs to the information for the example shown. By combining method-level change context, fine-grained code changes, commit messages, and application descriptions, we provide the LLM

with a multi-faceted view of the software changes. This approach results in more comprehensive, contextually aware, and user-focused release notes.

2.3 Adaptive Semantics Augmentation (Phase 3)

With multi-granularity information available, we design the LLM prompt to ensure accurate interpretation and desired summarization. We leverage the LLM's in-context learning ability by adding exemplars to each prompt, a technique proven effective in code summarization and generation tasks. In Figure 4, the top area (below system message, above code) shows an example.

Our exemplar curation process covers various code change categories, including bug fixes, new features, and refactoring. This approach enables LLM to learn different types of code changes and generate appropriate summarizations. For instance, when encountering minor refactoring changes unrelated to functionality, it can produce an empty summarization based on learned exemplars.

We implement category-wise adaptive exemplar selection to accommodate the diverse behaviors and requirements of different app types. This approach ensures the LLM receives contextually relevant examples, enhancing its ability to understand and summarize code changes accurately. For a Reading app, we select exemplars from other Reading apps within the dataset, providing similar feature types and code structures for more applicable and understandable release note generation.

Our exemplars cover a comprehensive range of code modifications. They include bug fixes (e.g., correcting a parsing error in an eBook format), new feature additions (e.g., implementing a dark mode for night-time reading), and refactoring efforts (like improving code maintainability through code reorganization). Based on previous studies [38], we prioritize "New Feature" and "Bug fix" exemplars, which are more frequent in Android app evolution. We also include exemplars showing refactoring changes with empty outputs to guide the LLM to rule out refactoring interference.

2.4 Release Note Synthesis (Phase 4)

After generating individual summarizations for a release, we synthesize them into cohesive and comprehensive release notes, addressing three key challenges. First, we tackle the lack of holistic context in individual summarizations, which can lead to redundancy or conflicts. Our synthesis process integrates these pieces, eliminating duplicates and presenting a unified view of changes. Second, we address the issue of overly technical LLM-generated summarizations. The synthesis transforms these into higher-level descriptions, highlighting key updates and improvements for non-technical users, or removes unnecessary low-level details. Lastly, we improve the logical organization of the content. LLM-generated summarizations may lack a coherent order, so our synthesis process structures the content, grouping related changes and presenting them in a logical sequence. This comprehensive approach ensures the final release notes are user-friendly, accurate, and logically organized, significantly enhancing their overall effectiveness and readability.

3 Evaluation

Our evaluation of VerLog is guided by following questions:

RQ1 *How effective is VerLog in generating accurate and comprehensive release notes?*

RQ2 *How does VerLog's performance compare to other baselines in release note generation?*

RQ3 *What is the impact of VerLog's individual components on its overall performance?*

RQ4 *How efficient is VerLog in terms of computational resources and generation time?*

RQ5 *How well does VerLog perform in real-world scenarios as evaluated by software developers?*

To answer these questions, we first describe the dataset we used for evaluation, metrics, and the setup. After that, we will answer all of them.

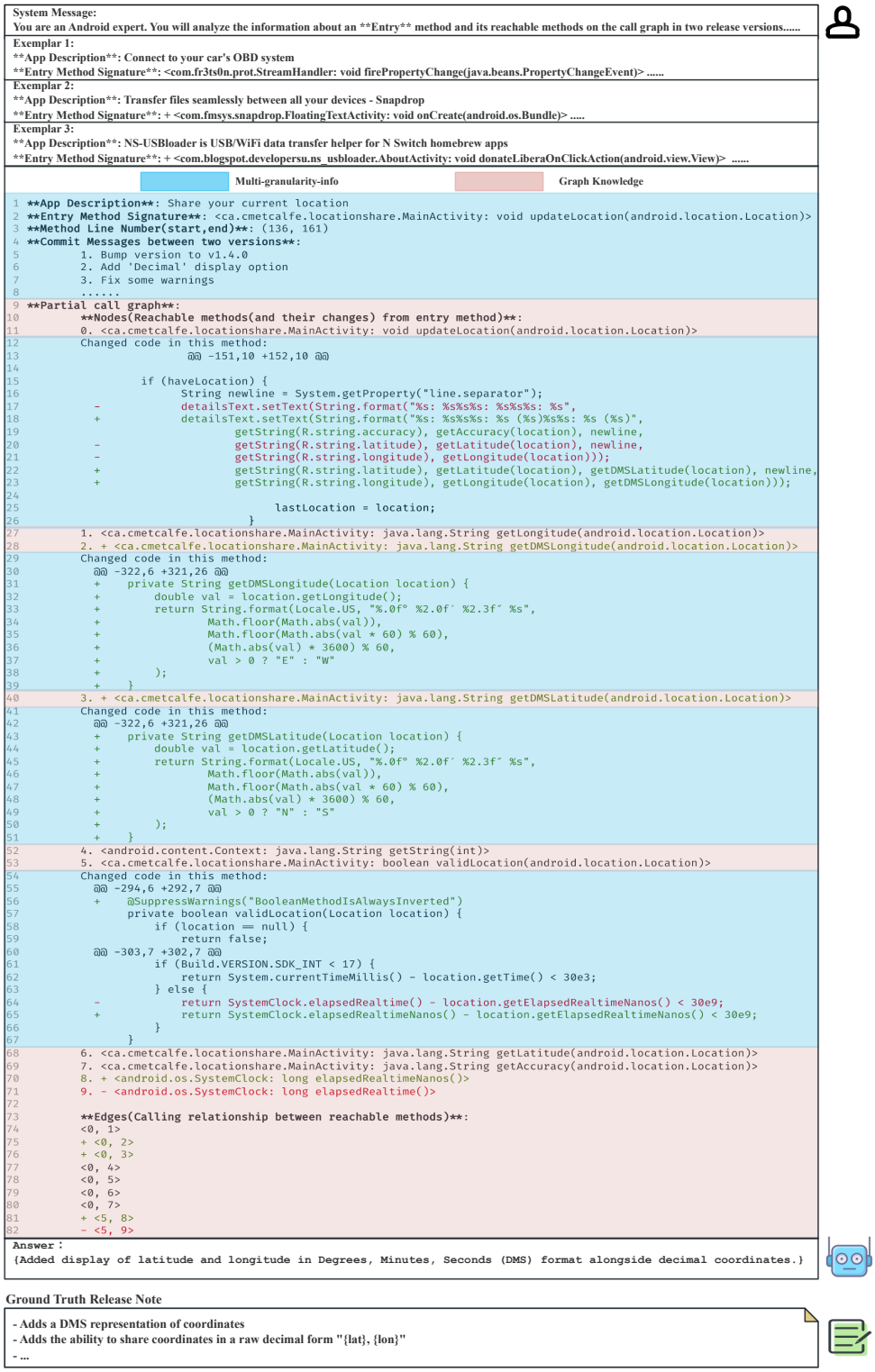

Fig. 4. An illustrating example of the complete prompting design including the LLM response in VerLog.

3.1 Experimental Setup

3.1.1 Dataset. The selection of our dataset was primarily driven by the need for high-quality release notes, which are crucial for a robust evaluation of our technique. We define high-quality release notes based on three key criteria: (1) *completeness*: comprehensive coverage of all significant changes including new features, bug fixes, and breaking changes; (2) *accuracy*: factually correct descriptions that align with code changes, commit messages, and related discussions; and (3) *readability*: well-structured, concise content that is easily understood by the target audience.

To establish reliable ground truth, we followed a systematic negotiated agreement procedure [5]. First, each author independently drafted the release note of each app using the above quality criteria while analyzing the same materials (app metadata and its source code, code diffs, commit messages, and related documentation). Then, via a consensus process, authors collaborated to merge their drafts and resolve any discrepancies, hence producing the final release notes as the ground truth.

Release note generation tools, including VerLog, typically require access to source code, repository metadata, and version histories. These artifacts are often unavailable in closed-source projects. On the other hand, the intended users of VerLog are developers who have access to their own codebases, whether proprietary or open-source. Therefore, we curated our evaluation dataset from F-droid [14], an open-source Android app repository, using a multi-stage selection process. We first filtered F-droid metadata for Java apps with `changeLog` field and GitHub-hosted repositories. This was followed by manual verification of URL validity and the presence of high-quality release notes in the app's history. We then downloaded and attempted to compile the selected apps to obtain unobfuscated APKs. The final dataset comprises 42 unique apps that were successfully built and processed, coming with 248 release notes—found in our artifact package.

3.1.2 Metrics. To evaluate VerLog's performance, we employed a combination of automated metrics and user studies, to cover both quantitative and qualitative evaluation.

During the dataset selection phase, we ensured that the ground truth release notes were easily verifiable, as mentioned previously. We then manually compared the VerLog-generated release notes with the ground truth. To minimize subjectivity, three authors as evaluators independently assessed each entry in the generated notes, making binary decisions (Match or No Match) based on semantic correspondence with ground truth entries. After independent assessment, we discussed any disagreements to reach a final consensus. Based on these manual consensus, we compute $\text{precision}=\frac{\text{Matched Entries}}{\text{Total Generated Entries}}$ and $\text{recall}=\frac{\text{Matched Entries}}{\text{Total Ground-Truth Entries}}$, and then F1 score$=2\times\frac{\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$— i.e., matched entries are true positive, whereas non-matches are false positive or false negative. This human-in-the-loop approach allowed for nuanced judgments that account for semantic equivalence, even when there are syntactic differences. It aligns more closely with user needs, as in practical scenarios, users of release notes are primarily concerned with whether the important changes and updates are accurately captured, rather than the specific phrasing used.

Traditional metrics like BLEU and ROUGE, while useful for many NLP tasks and used in related work [23] [35] [21], have limitations when applied to LLM-generated content, especially in specialized domains like software documentation. They primarily measure word or phrase overlap, which can be misleading when evaluating LLM outputs that often involve paraphrasing or reformulation. These metrics can unfairly penalize semantically correct but lexically different variations, which are common and often desirable in LLM-generated text. Moreover, they may not capture the nuanced accuracy required in technical documentation like release notes, where precise conveyance of information is more critical than exact wording.

For user study, to assess qualitative aspects, we conducted a user study where participants rated VerLog's output, baselines, and ground-truth on a 5-point Likert scale for: (1) *Overall Quality*, (2)

Completeness, (3) *Accuracy*, (4) *Readability*, and (5) *Preference*. This approach combines quantitative assessment of semantic accuracy with qualitative evaluation of usability and effectiveness.

3.1.3 Baselines. We selected five state-of-the-art (SOTA) baselines to demonstrate the merits of VerLog, as summarized and justified below.

(1) **TextRank**: A text summarization technique adapted for release note generation [34]. It uses commit messages and pull-request titles, employing the GloVe [39] word embedding for keyword extraction.
(2) **GitHub Changelog Generator (GCG)**: An industry-standard tool with over 7,400 GitHub stars, representing a widely-adopted solution to automated change-log creation [12].
(3) **DIFF**: A simple but potentially strong approach using LLMs to generate release notes from raw diff hunks. This baseline represents a pure code-summarization approach, comparable to recent pre-trained models for code-change tasks [26] [27] [15].
(4) **GPT-4o**: a general, powerful foundation LLM [37] that has the potential to directly serve software change documentation purposes.
(5) **CodeLlama**: Similar to the above but using a code-specific LLM [30] which might work better for coding tasks such as change documentation than general-purpose LLMs.

The overall baseline selection is further justified by that (1) they cover both traditional/non-LLM-based (i.e., the first two) and LLM-based (i.e., the other three) techniques, (2) DIFF represents approaches specialized for summarizing code changes, and (3) the last two baselines directly leverage both general-purpose and code-specific foundation LLMs, for which we chose `GPT-4o` released in May 2024 and `CodeLlama-70b-instruct`, respectively, as they were SOTA LLMs in their respective class by the time VerLog's base model was released.

For TextRank and GCG, we adopted their original/default settings. For the last two LLM baselines, `CodeLlama` and `GPT-4o`, we use the same multi-granularity information (i.e., app description, commit messages, and changed methods with detailed line-level changes) taken by VerLog as their input. We prompt the LLMs to generate release notes using the template in Figure 5. For DIFF, we use the same LLM underlying VerLog (i.e., GPT-4o-mini) for fair comparisons, and the same template but without providing app description and commit messages.

3.1.4 Environment. We ran all our experiments on a sever with Threadripper PRO 3995WX, four A6000 GPUs, and Ubuntu 20.04 as operating system. For the specific large language model used, we chose `ChatGPT-4o-mini` [36], which is among the most cost-efficient while reasonably intelligent.

3.2 Evaluation Procedure

For RQ1 (effectiveness), we evaluated VerLog's ability to generate accurate and comprehensive release notes using Precision, Recall, and F1 score. These metrics were computed by comparing the generated release notes with the ground truth, focusing on the semantic correspondence between entries rather than lexical similarity.

To address RQ2 (comparison to baselines), we applied the same metrics used in RQ1 to our baseline approaches which are configured as described above. For the shared types of information, all the baselines and VerLog are fed with the same inputs.

For RQ3 (ablation study), we created several variants of VerLog, each with a key component removed or modified. We evaluated these variants using the same metrics as in RQ1 and RQ2.

RQ4 (efficiency) was addressed by measuring three key aspects: token usage, time cost, and peak memory usage for generating release notes.

Finally, for RQ5 (in-the-wild evaluation), we conducted a user study with experienced software developers. In total 10 participants were asked to rate the automatically generated release notes on a 5-point Likert scale across several dimensions that cover what defines a high-quality release note.

3.3 Results and Analysis

3.3.1 RQ1: Effectiveness of VerLog. To evaluate the effectiveness of VerLog in generating release notes, we compare these generated notes against the ground truth. The reliability of our evaluation is demonstrated by strong inter-rater agreement scores before consensus building: Cohen's Kappa scores between pairs of raters are 0.7844 (Rater-1, Rater-2), 0.7376 (Rater-1, Rater-3), and 0.8320 (Rater-2, Rater-3), and Fleiss' Kappa score across all raters (0.7842) indicate substantial agreement.

```
# Role
You are an Android expert. You will analyze the information about changed methods between two release versions (marked as ‘v1‘ and ‘v2) of an Android app and then output a release note based on the analysis of changes.
# Input Content
1. App Description: [will be provided]
2. Method Signature: [will be provided]
3. Commit Messages Between Two Versions: [will be provided]
4. Changed Statements: [will be provided. The diff hunks of the changed user-defined method]
# Task
Summarize the functionality changes as a concise release note, focusing on end-user-perceivable features between ‘v1‘ and ‘v2‘ based on the code changes and other information provided above.
# Input Template
{App Description}
{Commit Messages}
{All Changed Methods with statements}
```

Fig. 5. Prompt template used for LLM-based baselines CodeLlama and GPT-4o.

As shown in Table 1, the effectiveness results demonstrate VerLog's strong performance in generating release notes. The precision of 0.7077 indicates that a significant majority of the entries in the generated release notes accurately correspond to actual changes in the software. This high precision suggests that VerLog is effective at avoiding the inclusion of irrelevant or incorrect information in its output. The recall of 0.8002 is particularly noteworthy, as it shows that VerLog successfully captures a large proportion of the relevant changes that should be included in the release notes. This high recall indicates that VerLog is comprehensive in its analysis, rarely missing important updates or modifications that users and developers would need to know about. However, for uncaptured but important changes, our analysis reveals that they often belong to non-code changes, e.g., UI changes. The F1 score of 0.7111, which provides a balanced measure of precision and recall, further confirms VerLog's overall effectiveness. This score suggests that VerLog achieves a good balance between including relevant information and avoiding unnecessary details. The high recall, in particular, is crucial for ensuring that users are informed about all significant changes, while the solid precision helps maintain the readability and relevance of the generated notes.

> *VerLog achieves precision of 0.71, recall of 0.80, and an F1 score of 0.71, indicating that it is capable of generating release notes that are both accurate and comprehensive.*

3.3.2 RQ2: Comparison with Baselines. The chosen baselines represent different approaches to automated release note generation, ranging from text summarization techniques to widely-used tools and both simple and advanced LLM applications. Table 1 shows the results.

Table 1. Effectiveness versus baselines

	Precision	Recall	F1
VerLog	**0.7077**	0.8002	**0.7111**
TextRank	0.3602	**0.8011**	0.4473
GCG	0.1731	0.1932	0.1909
DIFF	0.2499	0.5532	0.3166
CodeLlama	0.4220	0.5627	0.4300
GPT-4o	0.6002	0.6713	0.5874

For textual-based techniques, TextRank achieved an F1 score of 0.4473, significantly lower than VerLog's 0.7111. While TextRank shows high recall, comparable to VerLog's, its precision is considerably lower, suggesting it struggles to distill the most relevant information for release notes. GCG achieved the lowest scores across all metrics, indicating significant limitations in capturing nuanced changes and producing notes that align with human-written references.

Among LLM-based approaches, DIFF showed moderate performance with higher recall than precision, suggesting it captures many changes but struggles to focus on the most important ones. CodeLlama achieved moderate performance with balanced precision and recall, while GPT-4o showed stronger results with improved precision and recall. However, both still fall short of VerLog's performance, highlighting that merely applying powerful language models without proper code analysis and prompt engineering is insufficient for high-quality release note generation.

VerLog outperforms all baselines in all metrics, with particularly notable improvements in precision. The substantial advantages over these baselines, including advanced LLM-based approaches, underscores the effectiveness of our approach combing graph-based code analysis, multi-granularity

VerLog significantly outperforms all baselines in release note generation, achieving a 21% higher F1 than the best baseline (GPT-4o) and a 59% improvement over traditional approaches like TextRank, demonstrating the merits of combining graph-based code analysis with LLM capabilities.

Table 2. Description of ablation design

Ablation	Graph Knowledge (Phase 1)	Multi-granularity (Phase 2)	Exemplars (Phase 3)
Random-exemplars	✓	✓	○
Zero-Exemplars	✓	✓	×
Only-Code-Diff	×	✓	×
No-Code-Diff	✓	×	✓
No-Graph	×	✓	✓

3.3.3 RQ3: Ablation Studies. In this subsection we conducted ablation study to better understand the contribution of each component in VerLog. This study helps isolate the impact of individual design elements on the overall performance of our approach.

We considered four main ablations, which are shown in 2. They are: (1) **No-Graph**: Removes graph knowledge-based prompting (Phase 1). (2) **Zero-Exemplars**: Eliminates adaptive semantics augmentation (Phase 2), (3) **Random-Exemplars**: Uses random exemplar selection instead of adaptive selection.(4) **No-Code-Diff**: Removes multi-granularity feature composition (Phase 3), and (5) **Only-Code-Diff**: Retains only code diff information, removing Phases 1 and 2.

Our ablation experiments revealed several key findings. *Firstly: Graph knowledge-based prompting is crucial.* The No-Graph variant saw F1 score drop from 0.7111 to 0.6918, highlighting the value of semantic graph representation for code changes. Secondly, *Adaptive semantics augmentation is significant*: Zero-Exemplars (F1: 0.6714) and Random-Exemplars (F1: 0.7013) both underperformed compared to the full VerLog, demonstrating the importance of adaptive exemplar selection. Furthermore, *Multi-granularity feature composition is critical*: No-Code-Diff variant showed a dramatic performance drop (F1: 0.2571), emphasizing the need for diverse information sources in release note generation. The full VerLog system (F1: 0.7111) outperformed all ablated variants, including the Only-Code-Diff ablation (F1: 0.6354), validating the necessity of our multi-faceted approach.

Table 3. Results of ablation studies

Ablation	Precision	Recall	F1
Full Design	**0.7077**	**0.8002**	**0.7111**
Random-exemplars	0.6888	0.7511	0.7013
Zero-Exemplars	0.6499	0.7673	0.6714
Only-Code-Diff	0.5951	0.7498	0.6354
No-Code-Diff	0.2667	0.2848	0.2571
No-Graph	0.6756	0.7703	0.6918

These results confirm VerLog as a balanced system where each component plays a vital role. The graph-based representation provides semantic context, adaptive exemplar selection guides relevant output, and multi-granularity information ensures comprehensive coverage. Future work could explore refining exemplar selection strategies and integrating additional software development artifacts to further enhance performance.

Moreover, this ablation study offers insights into potential areas for future improvement. For instance, the relatively strong performance of the "Random-Exemplars" variant suggests that there might be room for further refinement in our exemplar selection strategy. Additionally, the significant impact of removing multi-granularity information points to the potential value of exploring even more diverse information sources or developing more sophisticated methods for integrating different types of software development artifacts.

All the key design elements in VerLog, graph-based prompting, adaptive exemplar selection, and multi-granularity feature composition, contribute significantly to its overall effectiveness.

3.3.4 RQ4: Cost-Efficiency. The cost structure in VerLog primarily stems from token usage in LLM interactions. Each prompt consists of three main components: system instruction, exemplars, and the prompt body. For input token usage analysis, we measured the average tokens per full prompt, together with minimum and maximum number of tokens in one prompt (including all three components) using a tokenizer, with a result of 2,217, 10,909, and 2,978, respectively. Our measurements show that most prompts fall within the 2K-4K token range, with a relatively low average. This efficient token usage is achieved via our careful prompt engineering designed to minimize input length while maintaining effectiveness, as discussed in Section 4.1.2.

Regarding output tokens, which have higher pricing, the costs are minimal since we specifically instruct the LLM to generate a concise, single-sentence release note entry per prompt. Given this controlled output format, output token costs can be considered negligible in the overall cost calculation. This allows for accurate cost projections for typical release note generation tasks.

Time efficiency is another crucial aspect of our cost analysis. We record the time cost of each phase and certain subtasks, as seen in Table 4. As Phase 2 and Phase 3 are simpler in implementation, compared to Phase 1, and do not rely on multiple LLM API calls, compared to Phase 4, we count them together for time cost computation. The time cost analysis reveals major dominants in VerLog's pipeline: call graph construction (Phase 1) and LLM processing (Phase 4), each consuming approximately 46% of the total execution time. This distribution indicates that the computational overhead is primarily split between static analysis and language model inference, with other components having minimal impact on overall runtime.

Table 4. Time cost (in seconds) breakdown

Phase/Task		Time	Percentage
Phase 1	*Changes Localization*	0.53	1.73%
	Call Graph Construction	14.33	46.80%
	CMG Construction	<0.01	<0.01%
Phase 2-3 (i.e., prompt formation)		1.54	5.03%
Phase 4 (i.e., LLM inference)		14.22	46.44%
Total		30.62	100.00%

Call graph construction time (14.33s on average) represents a dominant due to the need for static analysis. This cost item scales with project size and complexity, as the analysis must track method relationships across the entire codebase on the call graph. The similarly substantial time cost in Phase 4 (14.22s) is subject to the total amount of prompts generated, instead of the inference time cost for one single prompt; of course, the efficiency of the LLM itself is also a key factor.

Notably, other components like changes localization (0.53s) and CMG construction (<0.01s) have minimal impact on overall runtime. The multi-granularity feature composition and adaptive semantics augmentation (Phases 2-3) together consume only about 5% of the total execution time (1.54s), indicating efficient implementation of these novel components.

The average total execution time of 30.62 seconds per release suggests that VerLog can be practically integrated into development workflows without introducing significant delays.

VerLog is reasonably efficient for practical use, and the primary cost dominants are static code analysis and LLM inference. These costs can be paid off by the quality of release notes it generates.

3.3.5 RQ5: In-the-Wild User Study. To evaluate VerLog's performance in real-world scenarios and gather insights from practitioners, we conducted an in-the-wild user study. This study aimed to assess the quality and usability of release notes generated by VerLog compared to baseline approaches and existing low-quality release notes.

Methodology. We recruited 10 participants, all of whom were experienced Android developers and users. Each participant was assigned 10 different Android applications, specifically chosen for their historically low-quality release notes. For each application, we generated release notes using VerLog and our five baseline approaches. Participants were asked to evaluate each set of generated release notes across five key dimensions, rating them on a scale of 1 to 5 (where 1 is the

lowest and 5 is the highest): (1) *Overall Quality*, (2) *Completeness*, (3) *Accuracy*, (4) *Readability*, and (5) *Preference* (compared to existing low-quality release notes).

Table 5 reveals results of how participants evaluated different metrics on VerLog and baselines. VerLog achieved the highest overall average score of 3.99, but the distribution of scores across different metrics and approaches offers valuable insights into the strengths and limitations of various techniques. A clear pattern emerges when comparing LLM-based approaches (VerLog, DIFF, GPT-4o, CodeLlama) with traditional textual-based methods (TextRank, GCG). LLM-based approaches consistently received higher scores across all dimensions, particularly in readability and overall quality, suggesting that LLMs' natural language capabilities contribute significantly to generating comprehensible and well-structured release notes.

Table 5. User study results

	Overall Quality	Completeness	Accuracy	Readability	Preference	Average
VerLog	**4.00**	**3.81**	**3.63**	**4.56**	**3.94**	**3.99**
TextRank	2.94	3.44	3.50	2.25	2.38	2.90
GCG	2.25	2.31	3.13	2.69	2.19	2.51
DIFF	3.00	3.13	3.50	3.13	2.50	3.05
CodeLlama	3.12	3.26	3.35	3.65	3.03	3.28
GPT-4o	3.24	3.38	3.44	4.15	3.23	3.49

Textual-based approaches also showed notably lower scores in user preference. This indicates that while these approaches may capture relevant information, they struggle to present it in a user-friendly manner. GCG's consistently low scores across all metrics suggest that relying solely on repository metadata might be insufficient for generating quality release notes.

Notably, participants' preference scores aligned more closely with readability ratings than accuracy/completeness, suggesting that the clarity and presentation of information might be more important to users than technical precision/coverage. This observation is supported by VerLog and GPT-4o achieving the highest readability scores and correspondingly high preference ratings.

Other than these metrics, participants provided additional comments that shed light on VerLog's performance. Some participants offered suggestions for improvement, such as: *"It would be helpful to have more granular categorization of changes in some cases."* and *"For larger updates, a brief summary at the beginning would enhance quick comprehension."*

The results of the user study not only validate VerLog's performance metrics from our controlled experiments but also demonstrate its practical value to developers. The positive reception and constructive feedback from experienced practitioners underscore VerLog's potential to significantly enhance the release note generation process in real-world software development contexts.

4 Discussion

4.1 Scalability of VerLog

4.1.1 Graph Complexity vs. Cost. To stress test VerLog's scalability, we conducted additional experiments with larger apps. With the same methodology as for curating our main evaluation dataset (Section 3.1.1), including ground truth building and automatic compilation, we identified 5 large-scale apps across 15 releases, with compiled APK sizes ranging from 30MB to 90MB—the largest in our main dataset is 21MB. We examine the (1) graph complexity of these larger apps together with the main dataset and (2) VerLog's performance on releases of these larger apps.

Table 6. Graph complexity across all apps

Metric	Min	Max	Mean
Size of complete call graph	143	65,840	4,947.0
#CMGs	1	306	12.3
Total size of all CMGs	1	2,645	78.0
Size of one CMG	1	440	6.3

Graph complexity. Table 6 summarizes graph sizes (in terms of #edges) across the combined dataset. Although the complete call graphs can be extensive, our Contextualized Minimal Graphs (CMGs) remain quite manageable. To manage graph complexity effectively, VerLog employs a two-pronged approach as detailed in Section 2.1.

First, we only extracted the immediate callees of changed methods, limiting the input size for constructing CMGs. Second, when building CMGs with changed methods with callees, we prune the call graph by only keeping user-defined methods. This design in VerLog prevents the formation of deeply nested structures while preserving the semantic relationships (between changed code and its context) crucial for accurate automated release note generation.

Second, we implement a CMG/prompt merging strategy to handle cases in which changes are scattered across many simple CMGs. Our analysis shows that, on the 5 large apps, a significant portion (59.3%) of CMGs contain only a single node, which could lead to fragmented prompts hence inefficient LLM utilization. To address this, we merge these single-node CMGs based on class and package hierarchy, ensuring that each resulting prompt either contains multiple single-node CMGs from the same class/package or represents a unique isolated change. This optimization significantly reduces the number of required prompts—for the large apps, it reduces the maximum number of prompts from 306 to 104 while maintaining semantic coherence. The merging threshold (currently set to single-node CMGs) can be customized based on specific requirements, offering flexibility in balancing prompt count and complexity.

We further examined the correlation between graph complexity and VerLog's computational cost. Since call graph construction, slicing, and CMG construction all happen in Phase 1, for the cost here we consider the time taken in Phase 1. As seen in Table 7, while the complete call graph size shows a strong correlation with processing time, CMG-related metrics demonstrate no significant correlation. Specifically, the correlation coefficients for total #nodes in all CMGs (per release) and #nodes in one CMG indicate that our graph slicing approach effectively manages complexity irrespective of the original codebase size.

Table 7. Spearman coefficient between graph complexity metrics and VerLog's Phase 1 time cost and peak memory usage

Metric	Time Cost		Peak Memory	
	Coefficient	p-value	Coefficient	p-value
Size of complete call graph	0.916	<0.001	0.665	<0.001
#CMGs	0.034	0.561	0.053	0.362
Total size of all CMGs	-0.001	0.982	-0.029	0.624
Size of one CMG	-0.082	0.159	-0.149	0.010

Effectiveness. Moreover, on these larger apps VerLog also maintained strong effectiveness, with a precision of 0.91, recall of 0.71, and F1 score of 0.80, even higher than those on the main dataset. While these improved metrics may be partially attributed to the smaller number of samples and moderate changes in these releases, they demonstrate VerLog's ability to handle larger codebases effectively.

Cost. Results on resource utilization against these larger apps are shown in Table 8. The average peak memory usage of 3,667.67MB and time cost of 39.78 seconds in Phase 1 indicate that graph construction and analysis, while more resource-consuming for larger codebases, maintains practical scalability. This processing time primarily stems from the comprehensive static analysis required for call graph construction, which scales with codebase size but remains manageable through our optimized graph slicing approach.

Table 8. VerLog cost on large apps

Metric	Value
Avg. peak memory usage (MB)	3,667.67
Avg. (Phase-1) time cost (s)	39.78
Avg. token usage per prompt	3,751
Max. token usage per prompt	51,051
Avg. token usage per release	126,784

The average token usage per prompt (3,751) demonstrates that most prompts remain short, even for large apps. However, the maximum token usage of 51,051 tokens per prompt indicates that some complex changes require more extensive context. The average token usage per release (126,784) reflects the cumulative processing of multiple prompts for a complete release note. Despite higher token counts, our prompt partitioning (as detailed in Section 4.1.2) and merging (as described above) strategies effectively manage these larger requirements while maintaining generation quality.

In summary, despite the relatively small sample size of this additional evaluation, the further results suggest that VerLog remains practically scalable and effective for large-scale applications.

4.1.2 Long Prompt Handling. A critical challenge to LLM-based systems lies in the handling of long inputs (prompts) to the LLMs. We address this challenge through both preventive design and remedial strategies when necessary.

Our design of VerLog, Phase 1 in particular, proactively minimizes prompt length through preventive mechanisms. It reduces the overall scope of analysis via slicing (the complete call graph), while minimizing the input context via the CMG abstraction. Moreover, since a prompt corresponds to a CMG, our effort to minimize CMG complexity also contributes to prompt reduction, as we discussed in Section 4.1.1. The prompt formation geared towards graph-based reasoning (e.g., each prompt is only responsible for generating part of the release note corresponding to one subgraph) further helps reduce the prompt length.

For cases in which prompts might still exceed input limits despite these optimizations, we implement a *prompt partitioning* strategy, formulated as a bin-packing problem solved with the first-fit-decreasing (FFD) algorithm. This approach treats changed statements as items and prompts as bins (with max #tokens per LLM API call as bin size), maintaining necessary context by preserving method signatures and multi-granularity information while distributing changed statements across partitions. While this partitioning may end with necessitating multiple LLM API calls instead of one, each partition carries complete context to ensure accurate generation. For smaller CMGs, we optimize token usage by merging "fragmented" prompts based on class or package hierarchy to the extent where no two prompts contain only single methods from the same class or package.

Our analysis of token distributions demonstrates the effectiveness of the preventive design. In our dataset, prompts range from 2,217 to 10,909 tokens, averaging 2,978 tokens. Most prompts (92.39%) fall within 2-4K tokens, with only 7.42% using 4-8K tokens and a mere 0.19% exceeding 8K tokens. This distribution indicates that our length minimization strategies successfully keep most prompts within manageable limits.

Table 9. Effectiveness on all benchmarks (ALL) and benchmarks with (one or more) long (#tokens>4K) prompts (LP).

Metric	VerLog		TextRank		GCG		DIFF		GPT-4o		CodeLlama	
	ALL	LP	ALL	LP	ALL	LP	ALL	LP	ALL	LP	ALL	LP
Precision	0.66	0.71	0.33	0.36	0.13	0.17	0.28	0.25	0.60	0.55	0.42	0.47
Recall	0.77	0.80	0.84	0.80	0.15	0.19	0.60	0.55	0.67	0.56	0.56	0.53
F1	0.71	0.68	0.44	0.45	0.13	0.19	0.35	0.32	0.57	0.53	0.43	0.43

To validate our remedial strategy's effectiveness, we conducted experiments with a **4K** token limit, common in deployed open-source models. As shown in Table 9, for the 7.42% of prompts exceeding this limit, our results show minimal effectiveness impact. Several factors contribute to this robustness: (1) when multiple prompts are needed to generate one release note, those prompts typically have varying lengths, (2) prompt partitioning usually yields only two sub-prompts, and (3) these long prompts are distributed across different releases, not concentrated for one release.

Expectedly, textual-based approaches like TextRank and GCG maintain consistent performance regardless of app size. This is because they operate on commit messages and pull requests, which are inherently shorter and more uniform in length compared to code-based prompts. However, this textual-only nature, while computationally simpler, limits their ability to capture detailed code-level changes, as evidenced by their overall lower effectiveness comparatively.

4.2 Technical Generalizability and Limitations

For generalizability concerns, even though we applied VerLog to Android projects, VerLog's core design isn't inherently tied to Java or Android-specific features. The fundamental requirements of our approach – including software repository metadata, parsed source code, and call graphs – are

common across various programming languages. The prompt format we use for the Large Language Model (LLM) would remain consistent across different programming languages. Therefore, in terms of design, VerLog is is also language-agnostic.

As for limitations, VerLog faces three primary technical limitations. First, it struggles with non-code changes, particularly UI modifications in Android apps. While UI changes significantly impact user experience, our current implementation does not capture these effectively. A potential solution involves using multimodal LLMs to analyze and describe UI changes through screenshot comparisons. However, this approach would require comprehensive GUI testing across various device configurations, increasing complexity and resource demands.

Lastly, while VerLog automates much of the release note generation process, it doesn't eliminate the need for human oversight. As indicated in our title, VerLog is designed to enhance the process of release note generation, not to fully substitute developers. The tool aims to provide developers with a high-quality, automated starting point, allowing them to focus on refining and customizing the release notes according to specific needs or preferences. Release notes often require human oversight for legal, marketing, or user-specific language, and VerLog significantly reduces the manual burden by automating the most tedious part of the process—sifting through commits and code changes to extract meaningful information.

4.3 Threats to Validity

Internal Threats. The primary internal validity threat lies in potential implementation errors in VerLog and our experimental scripts. To mitigate this risk, we conducted thorough code reviews of our implementations against manageable testing scenarios. We also employed extensive unit and integration testing to verify the correctness of individual components and the system as a whole. Large Language Models are prone to hallucination [48], which could lead to inconsistent and unreliable release note generation. To address this issue, we set the temperature of the LLM to zero in our experiments, encouraging more deterministic outputs. We generated multiple samples for each release note and computed consistency metrics across these samples.Also, Each experiment was run multiple times, and we only considered results that were consistent across runs. Despite these precautions, we cannot entirely rule out the possibility that developers using VerLog may occasionally encounter slight variations in the generated release notes or require additional iterations to achieve the same level of quality demonstrated in our evaluation.

External Threats. Our study primarily used open-source Java Android applications from F-droid. This focus may limit the generalizability of our results to closed-source software projects or applications in other programming languages. While F-droid is a widely used benchmark in software engineering research, the release notes and development practices in this dataset may not fully represent those found in all real-world scenarios. For LLM selection, We chose `GPT-4o-mini` for our experiments due to its balance of capability and cost-effectiveness. However, with the rapid evolution of LLMs, more advanced models may not necessarily show the same degree of improvement as presented in our study. In terms of project Diversity: The scale and complexity of projects in our dataset may not fully represent the entire spectrum found in industry, very large or highly complex projects might present challenges not encountered in our study.

5 Related Work

Several studies have investigated the importance and characteristics of release notes in software development. Bi et al. [8] conducted a comprehensive study on release note production and usage, analyzing over 32,000 release notes from 1,000 GitHub projects. Yang et al. [47] focused specifically on release note patterns in popular Google Play Store apps.

The core focus of our work aligns closely with automatic release note generation. Moreno et al. [31] proposed ARENA, an approach for automated generation of release notes that has been a significant baseline in the field. Nath and Roy [34] used extractive summarization techniques, specifically an improved TextRank algorithm, for generating release notes from commit messages and pull request titles. More recent approaches have leveraged advanced natural language processing techniques. Kamezawa et al. [23] introduced RNSum, a large-scale dataset for release note generation via commit log summarization, proposing class-wise extractive-then-abstractive approaches using transformer-based models like BART. Jiang et al. [21] presented DeepRelease, a language-agnostic approach for generating release notes from pull requests in open-source software.

While not directly focused on release note generation, several works in related areas have informed our approach. Dunlap et al. [13] also utilized git-diff hunks and in-context learning to exploit LLM to help understand security patches. Fan et al. [15] explored the capabilities of LLMs for code change-related tasks, which aligns with our use of LLMs in VerLog. Grund et al. [18] introduced CodeShovel for constructing method-level source code histories, a technique that informs our approach to code differencing. Bansal et al. [6] demonstrated the effectiveness of function call graph context encoding for neural source code summarization, which inspired our graph-based approach in VerLog. Recent works like Ahmed et al. [3] on semantic augmentation of language model prompts for code summarization, and Geng et al. [16] on few-shot comment generation via in-context learning, have influenced our approach to leveraging LLMs in VerLog. Finally, recent works on automatic commit message generation by Zhang et al. [49] and Li et al. [25], as well as identifying geographic feature variations in Android apps using LLMs by Guo et al. [20], provide valuable insights into summarizing code changes, which is relevant to our task.

6 Conclusion

We present VerLog, a novel approach to automated release note generation using LLMs. VerLog addresses the challenges of manual release note creation by combining static code analysis-guided LLM graph knowledge reasoning, adaptive LLM prompting informed by app functionality domain, and multi-granularity information integration in few-shot in-context learning.

Our evaluation on dozens of real-world Android apps demonstrates VerLog's superiority over academic and industrial state-of-the-art baselines in terms of completeness, accuracy, and readability. The key technical contributions of VerLog include: (1) Graph knowledge-based prompting for semantic code change representation, (2) Adaptive exemplar selection for domain-specific release note generation and (3) Multi-granularity feature composition for comprehensive change documentation. These innovations enable VerLog to capture nuanced code changes and generate contextually appropriate release notes, outperforming existing techniques that rely solely on textual artifacts or simple code diffs. While our results are promising, limitations such as language specificity and potential LLM hallucinations warrant further investigation. Future work should focus on expanding language support, integrating VerLog into CI/CD pipelines, and exploring personalization based on user roles.

7 Data Availability

We have released our code and datasets to facilitate relevant future research, as found at https://figshare.com/s/8450d527685bee8fd06a.

Acknowledgments

We thank the reviewers for insightful and constructive comments. This work was supported in part by the Open Technology Fund (OTF) under Grant B00236-1220-00, and in part by Office of Naval Research (ONR) under Grant N000142212111.

References

[1] 2022. kaputnikgo/pilfershushjammerm. https://github.com/kaputnikGo/PilferShushJammer

[2] Surafel Lemma Abebe, Nasir Ali, and Ahmed E. Hassan. 2016. An empirical study of software release notes. *Empirical Software Engineering* 21, 3 (June 2016), 1107–1142. doi:10.1007/s10664-015-9377-5

[3] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639183

[4] Mubashir Ali, M. Irtaza Nawaz Tarar, and Wasi Haider Butt. 2020. Automatic Release Notes Generation: A Systematic Literature Review. In *IEEE 23rd International Multitopic Conference (INMIC)*. Bahawalpur, Pakistan, 1–5. doi:10.1109/INMIC50486.2020.9318191

[5] Ron Artstein and Massimo Poesio. 2008. Inter-coder agreement for computational linguistics. *Computational linguistics* 34, 4 (2008), 555–596. doi:10.1162/coli.07-034-R2

[6] Aakash Bansal, Zachary Eberhart, Zachary Karas, Yu Huang, and Collin McMillan. 2023. Function Call Graph Context Encoding for Neural Source Code Summarization. *IEEE Transactions on Software Engineering* 49, 9 (September 2023), 4268–4281. doi:10.1109/TSE.2023.3279774

[7] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. 2024. Graph of Thoughts: Solving Elaborate Problems with Large Language Models. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 16 (March 2024), 17682–17690. doi:10.1609/aaai.v38i16.29720

[8] Tingting Bi, Xin Xia, David Lo, John Grundy, and Thomas Zimmermann. 2022. An Empirical Study of Release Note Production and Usage in Practice. *IEEE Transactions on Software Engineering* 48, 6 (June 2022), 1834–1852. doi:10.1109/TSE.2020.3038881

[9] Haipeng Cai. 2020. Embracing mobile app evolution via continuous ecosystem mining and characterization. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*. 31–35. https://doi.org/10.1145/3387905.3388612

[10] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A large-scale study of application incompatibilities in Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 216–227. doi:10.1145/3293882.3330564

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[12] GitHub Changelog Generator Contributors. 2024. GitHub Changelog Generator. https://github.com/github-changelog-generator/github-changelog-generator

[13] Trevor Dunlap, John Speed Meyers, Bradley Reaves, and William Enck. 2024. Pairing Security Advisories with Vulnerable Functions Using Open-Source LLMs. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 350–369. https://doi.org/10.1007/978-3-031-64171-8_18

[14] F-Droid.org. [n. d.]. F-Droid - free and Open source android app repository. https://f-droid.org/en/

[15] Lishui Fan, Jiakun Liu, Zhongxin Liu, David Lo, Xin Xia, and Shanping Li. 2024. Exploring the Capabilities of LLMs for Code Change Related Tasks. http://arxiv.org/abs/2407.02824 arXiv:2407.02824 [cs].

[16] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3608134

[17] Git SCM. 2024. Git Diff Documentation. https://git-scm.com/docs/git-diff.

[18] Felix Grund, Shaiful Alam Chowdhury, Nick C. Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing Method-Level Source Code Histories. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1510–1522. doi:10.1109/ICSE43902.2021.00135 ISSN: 1558-1225.

[19] Jiawei Guo, Xiaoqin Fu, Li Li, Tao Zhang, Mattia Fazzini, and Haipeng Cai. 2025. Characterizing Installation- and Run-Time Compatibility Issues in Android Benign Apps and Malware. *ACM Transactions on Software Engineering and Methodology* (2025). doi:10.1145/3725810

[20] Jiawei Guo, Yu Nong, Zhiqiang Lin, and Haipeng Cai. 2025. Code Speaks Louder: Exploring Security and Privacy Relevant Regional Variations in Mobile Applications. In *IEEE Symposium on Security and Privacy (S&P)*.

[21] Huaxi Jiang, Jie Zhu, Li Yang, Geng Liang, and Chun Zuo. 2021. DeepRelease: Language-agnostic Release Notes Generation from Pull Requests of Open-source Software. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. 101–110. doi:10.1109/APSEC53868.2021.00018 ISSN: 2640-0715.

[22] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 135–146. doi:10.1109/ASE.2017.8115626

[23] Hisashi Kamezawa, Noriki Nishida, Nobuyuki Shimizu, Takashi Miyazaki, and Hideki Nakayama. 2022. RNSum: A Large-Scale Dataset for Automatic Release Note Generation via Commit Logs Summarization. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 8718–8735. doi:10.18653/v1/2022.acl-long.597

[24] Jens Krinke. 2006. Effects of context on program slicing. *Journal of Systems and Software* 79, 9 (2006), 1249–1260.

[25] Jiawei Li, David Faragó, Christian Petrov, and Iftekhar Ahmed. 2024. Only diff Is Not Enough: Generating Commit Messages Leveraging Reasoning and Action of Large Language Model. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 745–766. doi:10.1145/3643760

[26] Bo Lin, Shangwen Wang, Zhongxin Liu, Yepang Liu, Xin Xia, and Xiaoguang Mao. 2023. CCT5: A Code-Change-Oriented Pre-trained Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco CA USA, 1509–1521. doi:10.1145/3611643.3616339

[27] Shangqing Liu, Yanzhou Li, Xiaofei Xie, and Yang Liu. 2023. CommitBART: A Large Pre-trained Model for GitHub Commits. http://arxiv.org/abs/2208.08100 arXiv:2208.08100 [cs].

[28] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic Generation of Pull Request Descriptions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 176–188. doi:10.1109/ASE.2019.00026 ISSN: 2643-1572.

[29] Paul W. McBurney and Collin McMillan. 2016. Automatic Source Code Summarization of Context for Java Methods. *IEEE Transactions on Software Engineering* 42, 2 (Feb. 2016), 103–119. doi:10.1109/TSE.2015.2465386

[30] Meta. 2024. Introducing Code Llama, a State-of-the-Art Large Language Model for Coding. https://ai.meta.com/blog/code-llama-large-language-model-coding/.

[31] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrian Marcus, and Gerardo Canfora. 2016. ARENA: an approach for the automated generation of release notes. *IEEE Transactions on Software Engineering* 43, 2 (2016), 106–127. doi:10.1109/TSE.2016.2591536

[32] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639187

[33] Sristy Sumana Nath and Banani Roy. 2021. Automatically Generating Release Notes with Content Classification Models. *International Journal of Software Engineering and Knowledge Engineering* 31 (Dec. 2021), 1721–1740. doi:10.1142/S0218194021400192

[34] Sristy Sumana Nath and Banani Roy. 2021. Automatically generating release notes with content classification models. *International Journal of Software Engineering and Knowledge Engineering* 31, 11n12 (2021), 1721–1740. https://doi.org/10.1142/S0218194021400192

[35] Sristy Sumana Nath and Banani Roy. 2021. Towards Automatically Generating Release Notes using Extractive Summarization Technique. 241–248. doi:10.18293/SEKE2021-119 arXiv:2204.05345 [cs].

[36] OpenAI. 2024. GPT-4o Mini: Advancing Cost-Efficient Intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[37] OpenAI. 2025. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/.

[38] Luca Pascarella, Franz-Xaver Geiger, Fabio Palomba, Dario Di Nucci, Ivano Malavolta, and Alberto Bacchelli. 2018. Self-reported activities of Android developers. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, Gothenburg Sweden, 144–155. doi:10.1145/3197231.3197251

[39] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[40] Václav Rajlich. 2014. Software evolution and maintenance. In *Future of Software Engineering Proceedings*. 133–144.

[41] Sristy Sumana Nath and Banani Roy. [n. d.]. Practitioners' expectations on automated release note generation techniques. *Journal of Software: Evolution and Process* n/a, n/a ([n. d.]), e2657. doi:10.1002/smr.2657 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/smr.2657.

[42] Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. 2024. GraphGPT: Graph Instruction Tuning for Large Language Models. http://arxiv.org/abs/2310.13023 arXiv:2310.13023 [cs].

[43] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What makes a good commit message?. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2389–2401. doi:10.1145/3510003.3510205

[44] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. The influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering*. 356–366. doi:10.1145/2568225.2568315

[45] Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. 2023. Can language models solve graph problems in natural language?. In *Proceedings of the 37th International Conference on Neural*

Information Processing Systems (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1345, 22 pages. https://dl.acm.org/doi/abs/10.5555/3666122.3667467

[46] Shengnan Wu, Yangfan Zhou, and Xin Wang. 2021. Exploring User Experience of Automatic Documentation Tools. In *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–6. doi:10.1145/3411763.3451606

[47] Aidan Z. H. Yang, Safwat Hassan, Ying Zou, and Ahmed E. Hassan. 2022. An empirical study on release notes patterns of popular apps in the Google Play Store. *Empirical Software Engineering* 27, 2 (March 2022), 55. doi:10.1007/s10664-021-10086-2

[48] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, et al. 2023. Siren's song in the AI ocean: a survey on hallucination in large language models. *arXiv preprint arXiv:2309.01219* (2023).

[49] Yuxia Zhang, Zhiqing Qiu, Klaas-Jan Stol, Wenhui Zhu, Jiaxin Zhu, Yingchen Tian, and Hui Liu. 2024. Automatic Commit Message Generation: A Critical Review and Directions for Future Work. *IEEE Transactions on Software Engineering* 50, 4 (April 2024), 816–835. doi:10.1109/TSE.2024.3364675

[50] Ziyi Zhang and Haipeng Cai. 2019. A look into developer intentions for app compatibility in android. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 40–44. doi:10.1109/MOBILESoft.2019.00016

[51] Thomas Zimmermann, Rahul Premraj, Jonathan Sillito, and Silvia Breu. 2009. Improving bug tracking systems. In *Proceedings of the IEEE/ACM International Conference on Software Engineering-Companion Volume*. 247–250. doi:10.1109/ICSE-COMPANION.2009.5070993

Received 2025-02-14; accepted 2025-03-31